

PROGRAMMING EXPERT

FAQ

Q What is meant by 'exception handling'?

A When you write a program, you tend to focus on what happens when everything works: opening a file, reading data, processing it and so on. This is the normal or 'unexceptional' flow of control. If something disrupts the expected flow the program will crash unless it has code to handle the problem. Such problems are called exceptions and the code that deals with one is an exception handler.

Modern languages make exception handling easy. In C# you can write instructions such as `try{open file}catch{exception handler}`. If the file opens, the catch clause is ignored. If it doesn't, the exception handler is executed. A bullet-proof program will have a catch clause to handle any failure for every instruction, but in practice this is usually too much work.

Mike James

IT author, lecturer and programmer

mike@computersshopper.co.uk

Useful screensavers

To stop your computer wasting time while you're not using it, Mike James shows you how to create a useful screensaver



Screensavers are no longer necessary to stop a static picture burning itself permanently on to your display. However, they are a useful way of automatically running a program. If you don't press a key or move the mouse for a set time, the screensaver kicks in. This provides us with a way of running a program when the computer is idle – and, perhaps, presenting a report for when you return to work.

What a screensaver does once loaded is up to you. A screensaver will also run even before you have logged on to a PC, meaning there's great scope for clever programming. So screensavers aren't a complete waste of time. They might well be just the opposite: a way of using otherwise wasted computing time.

So how do you create a screensaver application? It is surprisingly easy, though you might not think so if you look it up in the documentation. There have been various attempts over the years to create a screensaver framework. Each has generally made something simple seem more difficult. Visual C# 2005 Express Edition, for example, has a Screen Saver Starter Kit project type. This creates a screensaver for you without any effort but, like many Microsoft-supplied examples, it's too sophisticated and does too much to make the basic principles obvious. It shows you how to create a traditional screensaver rather than helping you do something creative. It also fails to explain how to implement a screensaver preview. This month's project aims to correct these omissions.

SAVE OUR SCREENS

We'll start with a really simple example of a project to see how things work. Although screensavers seem special, they are just standard EXE files with their filenames changed to end in SCR. They live in either the Windows/System or Windows/System32 directory. When you use the Desktop Properties dialog box to choose a screensaver, all that happens is that Windows scans the relevant directories for all files ending in SCR and lists them as potential screensavers.

When the computer has been idle for a specified amount of time, in simple terms the selected screensaver is loaded and run just as a standard EXE file would be. There's a little more to it than this, but for now the only additional thing we need to take into

account when writing our first screensaver is that the screensaver loader passes the screensaver a command-line parameter to tell it what to do.

PARAMETER	MEANING
/s	Start in screensaver mode
/c	Show configuration dialog box with whatever window is currently active as the parent window
/p	Show a preview of your screensaver
None	Show the configuration dialog box with no parent window

The table shows what command-line parameters Windows may pass to your screensaver. The good news is that you only need to support the /s option to create the simplest possible screensaver.

Start a new Console project called ScreenSaver1. This provides you with the basic code you need: a Main function that is run when the application starts. This handles the command-line parameters as a single string, which is called args for arguments by default:

```
static void Main(string[] args)
{
}
```

If there are any command-line parameters, the args string will have a length greater than zero:

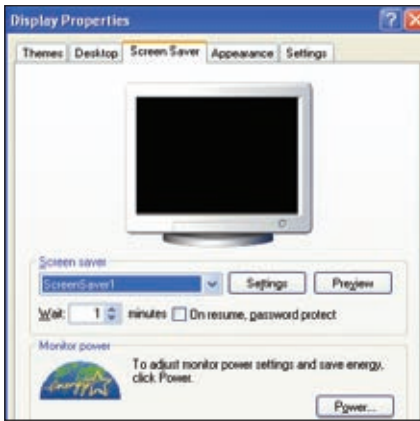
```
if (args.Length > 0)
{
}
```

All the valid parameters can be represented by two characters in lower case, so let's apply that rule:

```
string arg = args[0].ToLower()
.Trim().Substring(0, 2);
```

The result can now be used to decide what to do. You could use nested If statements but a Switch command (Select Case in VB) is neater and easier to follow:

```
switch (arg)
{
case "/c":
```



▲ Select your screensaver in Display Properties

```
//configured
break;
case "/s":
    ShowScreenSaver();
    break;
case "/p":
    //preview
    break;
}
}
else
{
    ShowScreenSaver();
}
}
```

The final else clause belongs to the initial if statement and simply shows the screensaver if there are no command-line parameters.

At some point in the future, we will have to supply functions to deal with all three possible command-line parameters, but for the moment all we need is the ShowScreenSaver function. In principle this could do anything you want, but the normal action of a screensaver is to load graphics to fill the screen. The simplest way of doing this is to load a form and customise it to fill the screen. For this simple example, we'll just load a form and leave the customisation for later. Add a form called screen to the project, there is no need to do anything to it. Just after the Main function defined above, add:

```
static void ShowScreenSaver()
{
    Application.Run(new screen());
}
```

This uses the Application object to run a new instance of the form on a new thread. This not only loads the form but starts its message pump running, which makes it responsive. After this, the thread running the main function can terminate and the screen form will remain active until the user closes it. Finally, to make this work we need to add the following to the start of the file continuing the Main function:

```
using System.Windows.Forms;
```

We are then ready to try the screensaver, but there is one small complication. At the moment the project is designated as a console project, which means it displays a command window and refuses to display Windows forms. We chose the console project just to get the system to generate the main function and other related

code. Now we want to turn it into a Windows application. To do this, select the Project, Properties menu item and change the Output type from Console Application to Windows Application. Set the Startup object to be the ScreenSaver1.Program to ensure that it is the main function that starts running first.

Run the program to check that the screen form appears. Next, you install the screensaver. To do this you first have to navigate to the project's debug directory, find ScreenSaver1.exe and rename it ScreenSaver1.scr.

There are two ways to activate the screensaver. You can copy it to the Windows/System32 directory, right-click on the desktop, select the Screen Saver tab and finally select ScreenSaver1 in the drop-down list. This method makes your screensaver available in the drop-down Screen Saver list thereafter.

A quick alternative method when you are just testing the screensaver is to right-click on the ScreenSaver1.scr file and select Install from the drop-down menu. This starts the Windows Screen Saver Dialog Box, which you can use to select your screensaver. It doesn't add your screensaver to the Screen Saver drop-down list. You can click the Preview button to see what the screensaver actually does. If you want to check that it starts after a period of inactivity, change the Wait time to one minute.

When your screensaver starts, the Screen form appears. If you move your mouse, nothing happens to the form. To close it, you must use the close box. The screensaver mechanism is a good way of getting almost any program started during a period of inactivity, and doesn't have to stop running as soon as the user does something.

A FORM-BASED SCREENSAVER

This is the traditional approach that you will find in the documentation and examples. Sometimes it's a good way of implementing a screensaver but if you are planning to open a form, there is a better way to build an application. Start a new Windows form project called ScreenSaver2. The problem with using a Windows form project is that it won't support command-line parameters by default. Contrary to popular opinion, though, adding support is fairly easy. If you open Program.cs, you will see the form application's main function and you can change this so that it receives command-line parameters thus:

```
static void Main(string[] args)
{
```

The form is then loaded using the command:

```
Application.Run(new Form1());
```

Unfortunately, we can't pass the form's constructor the command-line parameters in args. The solution is simply to define a new constructor that does accept the args array. Change the line that runs Form1 to:

```
Application.Run(new Form1(args));
```

Now we have to write the new constructor. Add to Form1.cs, following the default constructor Form1():

```
public Form1(string[] args)
{
    InitializeComponent();
```

```
m_args = args;
this.TopLevel = false;
}
```

Add the global variable:

```
private string[] m_args;
```

Any method in Form1 can access m_args to discover what the command-line parameters were. The TopLevel property is set to false to make the Form1 form invisible. You can't make the form loaded on startup invisible using Hide or by setting Visible = False because the Run method makes the form visible after the Load event handler is finished. With these changes, all the command-line argument processing can be performed in the Form Load event handler:

```
private void Form1_Load(
    object sender, EventArgs e)
{
    if (m_args.Length > 0)
    {
        string arg = m_args[0].ToLower()
            .Trim().Substring(0, 2);
        switch (arg)
        {
            case "/c":
                //configure
                break;
            case "/s":
                ShowScreenSaver();
                break;
            case "/p":
                //preview
                break;
        }
    }
    else
    {
        ShowScreenSaver();
    }
}
```

The ShowScreenSaver function is just:

```
void ShowScreenSaver()
{
    this.TopLevel = true;
}
```

BOUNCING TEXT

We need some graphics to show the sort of things a screensaver can do. As an example, we are going to bounce a message round the form. The simplest way of creating an animation is to use a timer, so place a timer control on the form and change the ShowScreenSaver to:

```
void ShowScreenSaver()
{
    this.TopLevel = true;
    this.DoubleBuffered = true;
    this.timer1.Interval = 20;
    this.timer1.Enabled = true;
}
```

This sets the timer to 20 milliseconds, which produces an update 50 times per second. Setting DoubleBuffered to true makes the animation smoother. We then have to write the Timer event handler to do all the work. First it gets the graphics object associated with the form so that it can use it to draw:

Introduction to Java Programming: Comprehensive Version

★★

£47

This is a huge book and must be a big seller, having made its sixth edition. It doesn't deserve such success. Author Daniel Liang has produced an academic book that attempts to be user friendly but fails. It starts simply enough but you won't get very far unless you are already a programmer.

The order in which topics are introduced isn't suitable for the beginner. Once we get beyond the basics of Java, we quickly move on to how to implement sorting algorithms and other topics of mainly academic importance. It doesn't cover using Java under Windows and it completely ignores the use of modern IDEs such as Eclipse or JBuilder as everything is done using the standard command prompt compiler.

There are some good things about this book. If you want a complete textbook-style discussion of Java it is a good choice. But if you are a complete beginner, want to use Java for fun or crave practical advice on creating Windows-oriented programs, look elsewhere.

SUMMARY

- ✓ Comprehensive; well produced
- ✗ Overly academic style

SUPPLIER www.amazon.co.uk
ISBN 0132221586
PAGES 1,301



Accelerated C# 2005

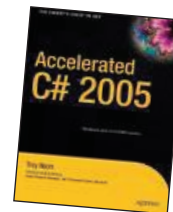
★★★★★

£19

If you already know, say, C++ or Java, you don't want a C# book that starts off with the basics. Trey Nash has attempted to provide a presentation of C# that is accelerated so you won't have to wade through the obvious stuff. It's a partial success, but a little too dry and reads like a manual in places.

My main gripe is that the book tries to explain every possible detail in a few sentences, producing a cloud of information. It works best in the discussions of some principle or other. For example, after explaining the mechanics of exceptions, a section on Achieving Exception Neutrality explains the philosophy behind exception handling and sets out what you should strive to do in your code. As a result, the best way to read this book is to skim-read the detailed introductions and focus on the more general advice. The final chapter, 'In search of C# canonical forms', is particularly valuable as it concentrates on how C# should be used. It probably should be turned into a book in its own right.

This book is recommended as long as you are prepared to skim-read some sections. The focus is on C# and how it is implemented in the CLR. You will need another book if you want to know how C# interacts with the .NET class library.



SUMMARY

- ✓ Short and direct
- ✗ Occasionally too much detail

SUPPLIER www.amazon.co.uk
ISBN 1590597176
PAGES 401

```
private void timer1_Tick(
    object sender, EventArgs e)
{
    Graphics g = this.CreateGraphics();
```

As will become clear, you can't assume that the screensaver is going to have a fixed-sized form on which to draw. Consequently, everything should be done in terms of a fraction of the available space. To do this, we first need to discover the size of the display area:

```
RectangleF bounds = this.
    ClientRectangle;
```

Next we can set the message to display and select a font size that is 1/20th of the height of the display area:

```
string displayM =
    "Buy Computer Shopper";
int fontsize = (int) bounds.Height
    / 20;
```

We need to set a font to use. Any font from the Arial family will do:

```
FontFamily fontFamily =
    new FontFamily("Arial");
Font font = new Font(
    fontFamily,
    fontsize,
    FontStyle.Regular,
    GraphicsUnit.Pixel);
```

To blank the message out at its current position, we can use the TextRenderer object and its MeasureText method. This is new in .NET 2.0 and returns a Size structure giving the height and width of the rectangle into which the text fits:

```
Size psize = new Size(int.MaxValue,
    int.MaxValue);
Size size = TextRenderer.
    MeasureText(g, displayM,
```

```
font, psize,
    TextFormatFlags.SingleLine);
```

The FillRectangle method can now be used to draw a rectangle of just the correct size in the current form's background colour:

```
SolidBrush b = new SolidBrush(this.
    BackColor);
g.FillRectangle(b, textpos.X,
    textpos.Y,
    size.Width,
    size.Height);
```

To move the text its position is updated. If it is about to leave the form, we reverse its direction of motion, bouncing it off the edge of the form:

```
textpos.X += v.X;
textpos.Y += v.Y;
if (textpos.X + size.Width >= bounds.
    Width
    || textpos.X <= 0) v.X = -v.X;
if (textpos.Y + size.Height >= bounds.
    Height
    || textpos.Y < 0) v.Y = -v.Y;
```

Finally, the text is drawn at its new position:

```
TextRenderer.DrawText(g, displayM,
    font, textpos,
    Color.Red);
g.Dispose();
}
```

We also need some additional global variables:

```
private Point textpos =
    new Point(0, 0);
private Point v =
    new Point(1, 1);
```

You should now see the message bounce its way around the form. If you resize the form it will automatically resize and bounce at the new edge.

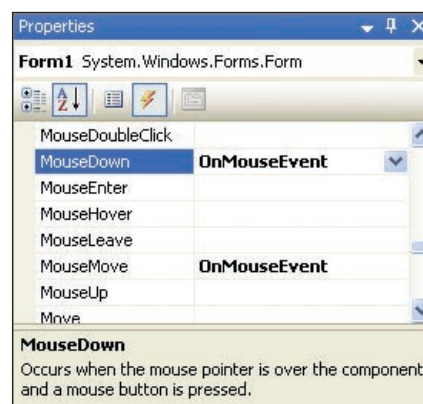
TAKING SHAPE

So far, we have a basic screensaver function but it doesn't look or behave like one. We have to resize the form to fill the screen and remove its border in the ShowScreenSaver function:

```
this.FormBorderStyle =
    FormBorderStyle.None;
this.TopLevel = true;
this.WindowState =
    FormWindowState.Maximized;
this.Capture = true;
Cursor.Hide();
```

To make the screensaver close when the mouse moves or if its button is clicked, we need a simple event handler:

```
private void OnMouseEvent(
    object sender,
    System.Windows.Forms.MouseEventArgs
    e)
{
    if (!MousePos.IsEmpty)
    {
        if (MousePos != new Point(e.X,
            e.Y))
```



▲ Setting the event handler in Properties


```

        this.Close();
    if (e.Clicks > 0)
        this.Close();
    }
    MousePos = new Point(e.X, e.Y);
}

```

We need another global variable to store the mouse position:

```
private Point MousePos;
```

This single event handler can deal with both the MouseMove and MouseDown events. View the form in design mode and use Properties to set both events to the OnMouseEvent function.

Now when you run the program it should fill the screen and close when the mouse moves or is clicked. You can improve the way it works by adding code to close it only if the mouse moves more than a set amount. You can also trap keyboard events to close the program.

PREVIEWING SCREENSAVERS

The screensaver should also run in preview or configuration mode in response to the switches /p and /c. The configuration mode is easy to implement and is described in the starter kit. The /p mode is more difficult and not described anywhere, so this is the next puzzle to solve.

When the /p option is present, the windows handle (hwnd) of the preview window is also passed on the command line as a sequence of decimal digits. For example, if the command line is /p 8324238, the handle of the preview window is 8324238. You could simply retrieve the hwnd and use it to construct a graphics object to let you draw on the preview window. This works, but there is a problem with doing things this way. The system expects the screensaver to attach a child window to the preview window and to draw on the child, not the preview. When the system needs to stop the screensaver for any reason, it simply closes the child window.

If you draw directly on the preview window, there's no child window to close and your screensaver will not be terminated when the user either previews another screensaver or selects a full-screen preview. You'll end up with lots of copies of your screensaver running at the same time, all trying to draw to the preview window. Ultimately the system will crash.

There seems to be no way of making a window a child of another window using the .NET framework, so once again we have to resort to using the Windows API. First we need to change our switch statement to deal with the handle passed on the command line:

```

case "/p":
//preview
    if (m_args.Length > 1)
    {
        this.TopLevel = false;
        m_hwnd = (IntPtr)Int32.Parse(m_args[1]);
        ShowScreenSaver();
    }
    break;

```

We also need to add the global variable:

```
private IntPtr m_hwnd;
```

If m_hwnd has a value, we need to display a

preview rather than the full screensaver. To do this, the ShowScreenSaver function has to be modified to initialise the form either for full-screen or preview display:

```

void ShowScreenSaver()
{
    if (this.m_hwnd == IntPtr.Zero)
    {
        All of the code
        already in ShowScreenSaver
    }
}

```

The else part of the If statement handles the preview and uses a number of API calls, definitions of which are given later.

We need to make the form's window a child window of the preview window. To do this, we need to retrieve and modify the form's window style to make it a child window:

```

else
{
    int style = GetWindowLong(
        this.Handle, GWL_STYLE);
    style = style | WS_CHILD;
    SetWindowLong(this.Handle,
        GWL_STYLE, style);
}

```

Next we tell the system that the preview window is the parent of the form's window and tell the form's window that its parent is the preview window:

```

SetParent(this.Handle,
    m_hwnd);
SetWindowLong(this.Handle,
    GWL_HWNDPARENT, (int) m_hwnd);

```

Now everything is set up for the form to display within the preview window, except that it's the wrong size and still has a border. To resize it, we need to know the size of the preview window:

```

this.FormBorderStyle =
    FormBorderStyle.None;
RECT r=new RECT();
GetClientRect(m_hwnd, out r);

```

To display the window, a final API call is used to resize the form's window and place it in front of the preview window:

```

SetWindowPos(this.Handle,
    HWND_TOP, 0, 0,
    r.Right, r.Bottom,
    SWP_NOZORDER |
    SWP_NOACTIVATE | SWP_SHOWWINDOW);
}

```

For all of this to work we need to add the following at the start of the file:

```
using System.Runtime.InteropServices;
```

We also need the following declarations, to make the API functions accessible, set up the RECT structure and define some constants:

```

public struct RECT
{
    public int Left;
    public int Top;
    public int Right;
    public int Bottom;
}

```

```

}
[DllImport("user32.dll")]
static extern bool GetClientRect(
    IntPtr hWnd,
    out RECT lpRect);

[DllImport("user32.dll",
    SetLastError = true)]
static extern int GetWindowLong(
    IntPtr hWnd,
    int nIndex);

const int GWL_STYLE = (-16);
const int WS_CHILD = 0x40000000;

[DllImport("user32.dll")]
static extern int SetWindowLong(
    IntPtr hWnd,
    int nIndex,
    int dwNewLong);

[DllImport("user32.dll")]
static extern IntPtr SetParent(
    IntPtr hWndChild,
    IntPtr hWndNewParent);

const int GWL_HWNDPARENT = (-8);

[DllImport("user32.dll")]
static extern bool SetWindowPos(
    IntPtr hWnd,
    IntPtr hWndInsertAfter,
    int X, int Y,
    int cx, int cy,
    uint uFlags);

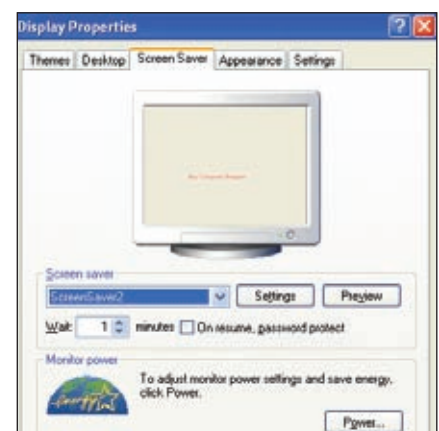
static readonly IntPtr HWND_TOP
    = new IntPtr(0);

const UInt32 SWP_NOZORDER = 0x0004;
const UInt32 SWP_NOACTIVATE = 0x0010;
const UInt32 SWP_SHOWWINDOW = 0x0040;

```

The preview should now work using the same tick event handler as the main screensaver. To see it in action, you have to install the screensaver.

You could change the program to display different graphics in the tiny preview window to when running full screen. You could, for example, show a few lines of explanation of the purpose of your screensaver. All that remains is for you to ponder what you want your next screensaver to do. There's no easier way of using otherwise idle processor time. ☐



▲ The screensaver in Preview mode